8-1995

Polymorphic Type Inference in Scheme

Steven L. Jenkins
Iowa State University

Gary T. Leavens
Iowa State University

Polymorphic Type Inference in Scheme

Abstract
This paper presents a type-inference system for Scheme that is designed to be used by students in an introductory programming course. The major goal of the work is to present a simple type inference system that can be used by beginning students, yet is powerful enough to express the ideas of types, polymorphism, abstract data types, and higher-order procedures. The system also performs some rudimentary syntax checking. The system uses subtyping, but only in a primitive fashion. It has a type datum which is a supertype of all types, and a type poof which is a subtype of all types. It uses and-types (intersection types) to control the use of datum and to generate accurate but simple types.

Keywords
abstract data type, type inference, supertype abstraction

Disciplines
Programming Languages and Compilers | Systems Architecture

Polymorphic Type-Checking in Scheme

Steven L. Jenkins
TR#95-21
August 1995

Keywords: abstract data type, type inference, supertype abstraction.
1992 CR Categories: D.3.3 [*Programming Languages*] Language Constructs -- Abstract data types.

Submitted for publication.

Some funding for the project was provided by NSF grant CCR 9593168.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

POLYMORPHIC TYPE-CHECKING IN SCHEME

Steven L. Jenkins
April 27, 1995

Abstract

This paper presents a type-inference system for Scheme that is designed to be used by students in an introductory programming course. The major goal of the work is to present a simple type inference system that can be used by beginning students, yet is powerful enough to express the ideas of types, polymorphism, abstract data types (ADTs), and higher-order procedures. The system also performs some rudimentary syntax checking. The system uses subtyping, but only in a primitive fashion. It has a type *datum* which is a supertype of all types, and a type *poof* which is a subtype of all types. It uses *and-types* (intersection types) to control the use of *datum* and to generate accurate but simple types.

TABLE OF CONTENTS

1. INTRODUCTION

This paper presents a type inference system for Scheme that is designed to be used by students in the Introduction to Programming class (CS227) at Iowa State University. The system is designed to aid students in understanding types, as well as help them find simple syntax and type errors in their programs. The main focus is on developing a type inference program that can handle Abstract Data Types (ADTs) and polymorphic procedures. However, the type system developed can handle a large subset of Scheme and its ideas may be more widely applicable. Another goal for the system is to infer types that are simple enough for students to understand. This paper motivates the need for the system, explains the subset of Scheme handled, describes how the algorithm works, provides pointers to related work, and discusses some future directions the work might take.

1.1. Motivation

Understanding types can often help beginning students grasp how their procedures should behave. For example, students often do not understand the difference between returning a value and printing it to the terminal. This can be clarified by showing students the types of two similar procedures, one that returns a specific value, and one that simply prints it. Understanding types also helps students build larger systems: by seeing the type-interface between various parts, students can understand how the system as a whole works, and can begin to see the utility of programming in modules or with ADTs.

With these ideas in mind, the major goal of this project is to help students understand types; by type-checking their procedures, students can see if their procedures have the types that they should, and they can use types of procedures to see how different parts of a large program work together. This aids students in understanding their own programs, as well as in understanding types. Other type-inference systems have been developed for Scheme; for example, STYLE [Lin93] and Soft Scheme [WrC93]. However, the complexity of the types output by these systems is often daunting for beginners.

The facilities for building and using ADTs with these systems is also different from the approach used in the text used in CS227, *Scheme and the Art of Programming* [SpF89]. Thus, another goal is to aid students in understanding and using Abstract Data Types (ADTs). By using the type-checker, they can see if their code makes proper use of an ADT's selectors, constructors, and mutators. This is perhaps the most important use of the type-checker.

1.2. Brief description of the type inference system

The type inference system presented in this report provides several useful features to beginning (and advanced) programmers: it infers types for a large subset of possible programs; it has provision for type declarations, especially ADTs; it handles procedures of variable arity in a limited fashion; the system introduces a bottom type called *poof* and a supertype of all types, *datum*. Intersection types are also handled in a limited fashion.

The algorithm itself extends a well-known approach to typing (i.e., Hindley-Milner type system with polymorphic type inference [Hin69][Car87]). The system is also approachable by students; almost all of the code has been written using the same subset of Scheme that students in introductory classes learn and that the type inference system works for. Thus, students can read the code to see how such a system works, and they can expand it themselves, providing further insights into type inference systems.

2. TYPE SYSTEM

This chapter covers the type system used in this paper. The first section discusses the subset of Scheme that the type inference system operates over, and the second section presents the type inference rules.

2.1. Domain of the type inference system

Since the type-checking system is designed to be used by students, it has some unusual features. The emphasis is on presenting useful information to students; i.e., information the students can understand. Thus, the syntax for types is quite simple. Also, the system is more restrictive than the Scheme language itself; for example, there are facilities to prevent the use of nested **define**s, even though nested **define**s are permitted in Scheme. The motivation for this is that the text for the class, *Scheme and the Art of Programming* [SpF89] does not use nested **define**s in the first thirteen chapters, which is the portion of the text normally covered in CS227. In general, the syntax accepted by the type-checker is the subset of Scheme covered in the first thirteen chapters of [SpF89]; however, some extensions have been made to increase the usefulness of the program. For example, **define** declarations can be of the form **(define name (lambda (arg1 ...) body))**, and of the form **(define (name arg1 ...) body)**. This has been done primarily so that programs from *Structure and Interpretation of Computer Programs* [AS85] could be used to test the type-inference system developed.

Another restriction is that intermediate values in a **begin** expression must have type *void*. This is because students often do not intend to use an implicit **begin**, yet they often do so in a **cond** expression. The last line of the following shows a fairly typical error.

```
(define add1-all
  (lambda (ls)
    (cond
     ((null? ls) '())
     ((pair? (car ls)) (add1-all (car ls)) (add1-all (cdr ls)))
     (else (add1 (car ls)) (add1-all (cdr ls))))))
```

In this procedure, the student, perhaps, has thought that **add1** has a side effect of incrementing the item it is applied to. Thus, the student thinks once **add1** has been applied to an element, it is sufficient to move to the next item in the list. By warning of intermediate expressions in an implicit begin having a non-*void* return type, this type of error can be prevented.

A restriction placed on **cond** and **if** expressions is that all of the test arguments must have type *boolean*. Hence, LISP programmers used to such programs as **member**, which returns **#f** or a value, will need to limit their use of symbols as booleans.

Another restriction is that students cannot define procedures that take a variable number of arguments (i.e., cannot define procedures using unrestricted lambda). However, students can use procedures that have variable arity as long as the procedures and their types are known to the type-inference program. Inferring types for procedures with variable arity could be handled as described in [DH94]. This restriction also implies that students cannot define procedures with rest parameters.

Functions that are used for their side-effects must have return types of *void*; for example, the procedure **for-each** requires that its first argument be a procedure that returns *void*.

The type system also makes no provisions for file I/O. This is not a great restriction to students as file I/O is beyond the scope of the introductory course, and in the first release of the code, types will be provided in the system. Other concepts covered in later chapters of [SpF89] have received only sporadic coverage. For example, **call-cc** is included in the type system, yet **with-input-from-file** is not. Completing the coverage of [SpF89] and, eventually, all of Scheme, as defined in [CR91] would make this work more widely useful.

These restrictions are made, both to simplify the type-checker, as well as to help simplify students' code. While these restrictions are not part of the Scheme language itself, we believe that students produce better code when obeying these restrictions.

2.2. Type information

This paper represents type information according to the following grammar:

```
<type>  ::==  number | boolean | string | character | symbol | void | datum | poof
            |  (-> (<type>*) <type>)
            |  (-> (<type> ...> <type>)
            |  (pair <type> <type>)
            |  (list <type>)
            |  (vector <type>)
            |  <type-variable>
            |  (and <type>+)

<type-variable> ::== A | B | C | D | ... | Z|?1|?2|...
```

Some of the rules require more explanation:

1. Types of procedures are represented by a list of three elements: an arrow (either '→ or '->, depending on the context), a list containing the types of the arguments to the procedure, and the return type of the procedure.

2. Complex types are represented as a list where the first item is the name of the complex type and the rest of the items are the primitive types in the complex, or container, type. The three complex types are lists, pairs, and vectors. For example, **(make-vector 10**

0) has type *(vector number)*, and **(cons 3 #t)** has type *(pair number boolean)*. These types can be nested, e.g., *(pair (list number) (pair number boolean))*.

3. Constrained quantification, for example, ∀a.(→ (list a) a), is represented by (-> ((list T)) T). Here, of course, the letter chosen to represent the type variable is arbitrary.

4. If an expression has more than one type, the type information is a list consisting of an *and*, followed by a list of the possible types. For example, **cons** has type

   (and (→ (S (list S)) (list S))
   (→ (T (list U)) (list datum))
   (→ (V W) (pair V W))

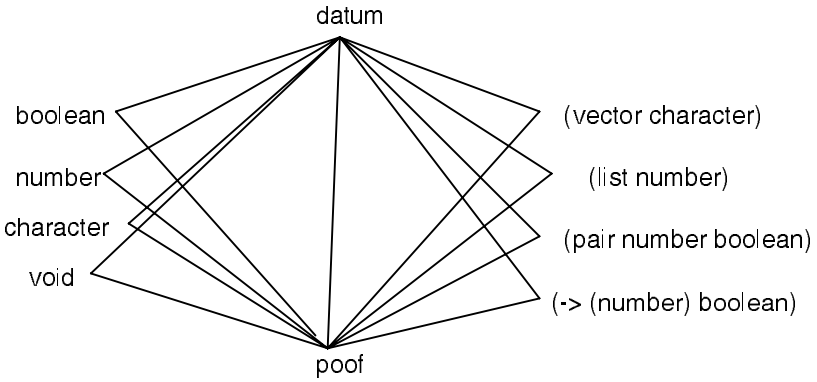
   Thus, **cons** creates all of these: homogenous lists, non-homogenous lists, and pairs. Procedures which have more than one type will be referred to as *and-types* within the rest of the paper. These are a form of intersection types as presented by [CD80].

To allow much of the power of Scheme without showing students very complex, if accurate, types, subtyping is used in a very restricted way. Two new types have been introduced: the type *poof*, and the type *datum*. The first is the return type of procedures that do not return to their caller. Such procedures do error handling or certain types of control flow; e.g., **error** and **call-cc**. The second is used for polymorphic procedures. For example, **cons** can have a return type of *(list datum)* if the second argument is not a list of the type of the first argument (i.e., **cons** is used with the type (→ (S (list T)) (list datum))).

A further extension to standard typing is the addition of procedures that can take an arbitrary number of arguments. These are handled by the ... (pronounced "dot-dot-dot") type. For example, the procedure + has type (→ (number ...) number), meaning that the procedure takes zero or more arguments, all of which are numbers, and returns a number. When combined with *datum*, this allows procedures like **writeln** to be given a type.

TYPE-INFERENCE RULES

The type-inference rules used here are quite simple and are based on Hindley-Milner type-system. To handle *poof* and *datum*, as well as a sub-typing notion, the following subtype relation holds: $\forall\, T,\ poof \leq T \leq datum$. The diagram below shows a graphical representation of the subtyping and supertype relationships.



Note that *list*, *pair*, *vector*, and $\rightarrow$ are type constructors. That is, an expression may have type *(list number)*, or *(pair number boolean)*, or even *(vector (pair number boolean))*. These types can be considered as container types, like container classes in an object-oriented paradigm.

The list of the type inference rules used is given below. Most of the notation follows Cardelli's presentation of type rules for his subset of ML [Car87]. For type environments, the expression $x{:}\tau$ is the binding of variable x to type τ. If Γ is a type environment, then $\Gamma.x{:}\tau$ is the same as Γ except x has type τ. If A and B are type environments, then $A \cup B$ is A with the types from B. The expression $A \vdash e{:}\ \tau$ means that given the type environment A, we can infer that e has type τ. The horizontal bar can be read as *implies*, where the top rule implies the bottom. The type-inference rules for the system are as follows.

[LAMBDA] $$\frac{\Gamma \cup \{x_1{:}\,\sigma_1, \ldots, x_n{:}\,\sigma_n\} \vdash e{:}\,\tau}{\Gamma \vdash (\mathit{lambda}\ (x_1 \ldots x_n)\ e){:}\ (\rightarrow\ (\sigma_1 \ldots \sigma_n)\ \ \tau)}$$

[BEGIN] $$\frac{\Gamma \vdash e{:}\,\tau, \Gamma \vdash c_i{:}\,\gamma_i, \forall i(\gamma_i\ \leq\ \mathit{void})}{\Gamma \vdash (\mathit{begin}\ c_1 \ldots e){:}\,\tau}$$

[APPL] $$\frac{\Gamma \vdash e_0{:}\left(\rightarrow\ (\sigma_1 \ldots \sigma_n)\ \tau\right),\ \ \forall 1 < i \leq n\ \Gamma \vdash e_i : \sigma_i}{\Gamma \vdash (e_0\ e_1 \ldots e_n){:}\,\tau}$$

[DATUM] $$\frac{\Gamma \vdash e{:}\,\tau}{\Gamma \vdash e{:}\ \mathit{datum}}$$ [POOF] $$\frac{\Gamma \vdash e{:}\ \mathit{poof}}{\Gamma \vdash e{:}\,\tau}$$

[IDE] $\Gamma.\ x{:}\,\tau \vdash x{:}\,\tau$

[IF1] $$\frac{\Gamma \vdash e_0{:}\,\mathit{boolean},\ \ e_1{:}\,\sigma,\ \ e_2{:}\,\sigma}{\Gamma \vdash \left(\mathit{if}\ e_0\ e_1\ e_2\right){:}\,\sigma}$$ [IF2] $$\frac{\Gamma \vdash e_0{:}\,\mathit{boolean},\ \ e_1{:}\,\mathit{void}}{\Gamma \vdash\ \left(\mathit{if}\ e_0\ e_1\right){:}\ \mathit{void}}$$

[LET] $$\frac{\Gamma' = \Gamma \cup \{x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n\},\quad \Gamma \vdash e_1{:}\,\tau_1, \ldots, \Gamma \vdash e_n{:}\,\tau_n,\quad \Gamma' \vdash e{:}\,\tau}{\Gamma \vdash (\mathit{let}\ ((x_1\ e_1) \ldots (x_n\ e_n))\ e){:}\,\tau}$$

[LETREC] $$\frac{\Gamma' = \Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}, \quad \Gamma' \vdash e_1 : \tau_1, \ldots, \Gamma' \vdash e_n : \tau_n, \quad \Gamma' \vdash e : \tau}{\Gamma \vdash (\mathit{letrec}\ ((x_1\ e_1) \ldots (x_n\ e_n))\ e) : \tau}$$

(Note: for all x_i, if they occur in a right-hand side expression, then they must be in a lambda expression.)

Note that there is no rule for **define**; this is because **define** is desugared into a **letrec** expression and then handled specially by the type inference system. The types *poof* and *datum* also require further discussion, as well as our use of *and-types* and procedures of variable arity. In general, these are restricted by not allowing the type system to infer expressions of these types. However, procedures of these types may be declared. This restriction enables us to do polymorphic type inference. Allowing procedures of variable arity to be declared lets us use the arithmetical procedures in a wide variety of settings.

3. TYPE INFERENCE ALGORITHM

The algorithm used to infer types is discussed in this section. It consists of three basic parts: desugaring of syntax, type-inference, and presentation sugaring. Each of these basic parts are defined separately. After presenting the basics of the algorithm, details of the interesting extensions to standard type inference are discussed in section 3.4.

3.1. Desugaring

To accomodate various types of procedural declarations, as well as simplify the type inference system and type table, several types of syntactic desugaring are done in the first stage.

All declarations with **define** are normalized to a standard syntax. For example, all procedures will look like

```
(define identity
   (lambda (x)
     x))
```

That is, a declaration of the form **(define (identity x) x))** will be translated into the above syntax. Once this has been done, the **define** expression is transformed into a **letrec** expression. For the **identity** procedure above, the **letrec** expression would be

```
(letrec ((identity (lambda (x) x)))
   identity)
```

By doing this transformation, the actual type inference algorithm only needs to deal with **letrec**, not any form of **define**. This is a great help in the complexity of the program. It also enables different styles of procedure declaration to be quickly added, modified, or deleted.

After the type for the transformation has been found, the global type environment is updated by binding the type of the **letrec**s body with the name of the procedure, so that any future reference to the name of the procedure will refer to the correct type.

Another desugaring that takes place is the translation of the procedures **caar**, **cadr**, **cddddr**, etc. procedure calls into the equivalent **car** and **cdr** combinations. This reduces the size of the type table by 28 entries, and provides greater flexibility. For example, if the instructor wishes to only allow lists, i.e., to disallow pairs, the only changes need to be made in the types of **car**, **cons**, and **cdr**, not to all 31 of the procedures that operate on lists.

Translation of **cond** expressions to the equivalent **if** expressions is done at this point. A flag can be set to require each **cond** expression to have an **else** condition, or not, as the user desires. This desugaring is only for simplicity of the type inference algorithm; fewer cases need to be handled once **cond** expressions are transformed into **if** expressions. No real efficiency gain is found, nor does it allow greater flexibility. This is merely done to reduce the complexity of the type inference algorithm itself.

Another desugaring that takes place is the translation of implicit **begin** expressions into explicit ones. This permits the handler for **begin** in the type inference algorithm to check for each intermediate expression returning a *void* type.

3.2. Type-Inference

As stated earlier, the algorithm used here is essentially the **j** algorithm from Milner's original paper on type inference [Mil78]. Given an expression, f, and an initially empty environment, p, type variables are instantiated for procedures and variables of unknown types. Then, once all variables are instantiated, unification is used to propogate constraints. If no solution to the set of constraints can be found, an error is signalled. A more complete description follows.

The algorithm only has ten cases: symbols, non-pairs (e.g. numbers, strings, vectors, etc.), **quote** expressions, **begin** expressions, **if** expressions, **lambda** expressions, **let** expressions, **letrec** expressions, and generic, or user-defined, expressions. Each of these are checked using the appropriate type-inference rules.

The most interesting case, of course, is the user-defined case. Here, the procedure being applied is checked to see if it has an entry in the global type table. If so, then its arguments are checked to make sure they match those given.

At this point, *and-type* expressions are taken care of by an ordered search. Thus, the first type given in the list of conjoined types that produces a usable unification is returned. Care must then be taken in entering new rules so that this property does not cause weak typing. For example, the type of **cons** is

```
(and (→ ?a (list ?a) (list ?a))
     (→ ?b (list ?c) (list datum))
     (→ ?d ?e (pair ?d ?e)))
```

The internal representation of types in the global type environment, ***global-var-types***, uses symbols constructed from ? and the lower-case letters. Thus, *?a* is a type variable, like *?T*, which is how the type variable might be printed out to the user. The first type that

is tested is *(→ ?a (list ?a) (list ?a))*, i.e., that **cons** is constructing a homogenous list. If that fails, then a unification is attempted to see if it is contructing a non-homogenous list. Finally, if both of the previous fail, the algorithm attempts to unify the actual arguments with the formal parameters *?d* and *?e*, thus the algorithm determines **cons** constructs a pair in such an expression.

Procedures with variable arity are then given a type that matches the length of the argument list. For example, **+** is given the type *(→ (number number number) number)* when it is called as in **(+ 3 4 5)**.

Once this has all been done, the bindings created by unification as in [Mil78] are all applied and, if no set of possible bindings exists, an error message is returned. If a possible set of bindings exists, then it is applied and the type is returned.

3.3. Re-sugaring

Once the type has been found, it is displayed (not returned) to the user. All type variables, which are of the form ?a, are rewritten as capital letters, e.g., T. This is simply an aid to the student in comprehending the types.

3.4. Implementation of type inference rules

This section describes the extensions to standard type inference that have been implemented by this system.

3.4.1. And-types

The system treats *and-types* by doing an ordered search over the possible types of a procedure. As mentioned in the section on the algorithm, the order of the rules is very important. Placing the most restrictive rule first and proceeding to the least restrictive will provide the strongest typing of an expression. In the example using **cons**, the rule could have been specified as follows.

```
(and (→ ?d ?e (pair ?d ?e))
     (→ ?a (list ?a) (list ?a))
     (→ ?b (list ?c) (list datum)))
```

In this case, however, the algorithm would always infer that **cons** creates pairs; it would never infer that **cons** creates lists. Thus, while still technically correct, the system would give a weaker and more complex type than is necessary. Therefore, care must be taken when creating *and-types* for the system.

3.4.2. Poof

The type *poof* is the subtype of all types. It is used for the types of procedures that do not return to their caller. In our system, it is only used to denote the return type of **error** and in the type of **call-cc**. For the purposes of CS227, though, **error** is important. The example below shows a typical use.

```
(define make-ratl
   (lambda (numr denr)
      (if (zero? denr)
         (error "The denominator cannot be zero!")
         (list numr denr))))
```

Here, the type of make-ratl should be *(→ (number number) (list number))* (or, more abstractly, *(→ (number number) ratl))*, but unless the type of error can be unified with *(list number)* and result in *(list number)*, or *ratl*, the type-checking will be incorrect. Since **error** is used in similar situations, and must always unify with other types, its return type, *poof*, must be the subtype of all types in Scheme. In the implementation, any occurrence of the type *poof* is automatically unified with any type variable.

3.4.3. Datum

In general, *datum* is the supertype of all Scheme types; all other types are subtypes of *datum*. However, it is well-known that having such a type can suppress detection of all type-errors because every expression could have type *datum*. The novel aspect of our program is its controlled use of *datum*. This is done by only allowing *datum* to be declared as part of a procedure's type, not inferred. With this in mind, only a few procedures should have type *datum*. Its primary uses are in building heterogenous lists (via **cons** or **list**) and producing output (via **writeln** or in **error**). One other use is in the procedure **make-vector**. If **make-vector** is passed a length as its only argument, then the vector created has undetermined fill values, hence the use of *datum*.

Within the algorithm, any other type or type variable unifies to type *datum*. Thus *number* and *datum* unify to *datum*. However, if *datum* is not one of the type variables, then normal unification takes place: *datum* is not introduced by the unification algorithm, but must be present from the type information stored in the global type table.

3.4.4. Procedures of Variable Arity

While procedures that take a variable number of arguments are handled somewhat, only procedures whose optional arguments are homogenous are handled; for example, the procedure **map-all** defined below can be typed, although its type cannot be inferred.

```
(define map-all
   (lambda args
      (cond ((null? args) '())
         ((null? (cdr args)) '())
         (else (apply map-all (list (car args)
                                    ((car args) (cadr args))
                                    (cddr args)))))))
```

While **map-all** can be typed, its type cannot be inferred in the present system; hence, procedures of variable arity are similar to *datum*. Currently, the only way for a procedure with variable arity to be typed is to use the procedure **deftype** to add an entry for **map-all** in the type table, unless the procedure is one of the built in procedures that have variable arity. The entry for **map-all** would be as follows: *(→ ((→ ?a ?b) ?a ...) (list ?b))*; i.e., **map-all** takes a procedure that operates on the arguments that follow after the procedure. Note that the optional arguments must all be the same type (even though that type might be datum).

Even though **map-all** can be typed, it cannot be type-checked. Using the current type-checker, an error would occur while attempting to translate the definition, as **lambda** must be followed immediately by a list of arguments, not by a name. One direction this research should take in the future is the addition of a type-inferencing system for these procedures. The work of Dzeng and Haynes looks particularly interesting for this [DH94].

3.4.5. ADTs

A major innovation of this project is producing a type-checker that will handle ADTs. Such a type-checker should allow the creation of new types, and be able to type-check the new types, their creators, selectors and mutators, as well as any procedures built using these types. Also, there must be a facility to handle information hiding, so that students will not be able to depend on a certain implementation for their ADTs. To see how these goals have been achieved, we will look at the ADT *ratl*, i.e., rationals. This is the first ADT students in the CS227 encounter in *Scheme and the Art of Programming*[SpF89].

The implementation for the constructors and selectors are shown below. Note that there are no mutators for this ADT.

```
(define make-ratl
   (lambda (numr denr)
      (if (zero? denr)
         (error "The denominator cannot be zero.")
         (list numr denr)))))

(define numr
   (lambda (ratl)
      (car ratl)))

(define denr
   (lambda (ratl)
      (cadr ratl)))
```

The types of this code can be inferred, and the types will be as follows:

- *make-ratl: (→ number number (list number))*
- *numr: (→ (list T) T)*
- *denr: (→ (list T) T)*

Note, however, that the '→ symbol will be represented by '-> in the actual program.

However, to hide an implementation, a file called **foo.def**, where **foo** is the name of the ADT, can be used. If a student is to use the file **foo.ss**, she can simply type **(type-check-file "foo.ss")** and the system will automatically look for a file **foo.def** that contains type information both for the representation of ADTs as well as the correct types for the ADTs procedures. At this point, the representation and the procedures' types will be stored in a table. Then the file **foo.ss** will be read in. Every procedure in **foo.ss** that has a corresponding entry in the table (from **foo.def**) must have a type that is unifiable with the type from the table using the given representation. For example, the contents of the file **ratl.def** are shown below.

```
(defrep ratl    (list number))
(deftype make-ratl (-> (number number) ratl))
(deftype numr (-> (ratl) number))
(deftype denr (-> (ratl) number))
```

Here, **defrep** marks that the implementation of a **ratl** is **(list number)**. Hence, any procedures in this file that are marked by a **deftype** should have all occurences of **ratl** replaced by **(list number)** in their actual implementation in the **ratl.ss** file. Thus, if the implementation of **numr** in **ratl.ss** is not of type *(-> ((list number)) number)*, then there is an mismatch between the specification and the implementation.

At this point in the research, every **.def** file must contain one and only one **defrep** expression. This is used to simplify checking the specification versus the implementation. It would significantly increase the complexity of this checking if more than one ADT were allowed to be specified in one file as coordinating the primitive procedures with their respective types would be more difficult. However, this restriction could be relaxed in future versions of this work if it is determined that the current system is needlessly restrictive.

3.5. Extending the system

Extending the system to accomodate all of R4RS compliant Scheme should not be too difficult. Much of the extension merely requires entries in the global type table. For example, the I/O procedures **open-input-file**, **open-output-file**, etc. merely require the appropriate types to be entered into the table. Other features, however, will require more difficult modifications. These include the aforementioned procedures of variable arity, as well as **define**s using rest parameters. Even though these additions will be more complex than simply entering types in a type table, they do not seem to be unrealistic. Dzeng and Haynes [DH94] have described a system that will solve the first problem. It only remains to be seen how difficult implementing their ideas into our framework will be. Also, implementing rest parameters should not be difficult once the framework for handling procedures of variable arity is in place. In short, the system seems to be flexible enough to handle further extensions without sacrificing its simplicity.

4. DISCUSSION

This section shows some sample student problems and demonstrates how the type inference system can help students solve some of their problems in understanding procedures. It also discusses the applicability of the system, and how effective it is in inferring types for code. The section also attempts to place this work in context with other works in developing type systems for Scheme. Specifically we look at STYLE and Soft Scheme, two approaches to type-checking Scheme. We then conclude the section by looking at future directions for this research.

4.1. Common Programming Problems

This section presents a brief discussion of programming problems students encounter. It is followed by an extended set of examples of interactions with Scheme interpreters and the type inference system.

1. attempt to take **car** or **cdr** of a non-list or non-pair
2. calling a procedure with the wrong number of arguments
3. a parenthesis problem
4. a problem with ADTs
5. a problem with **if**
6. a problem with **cond**
7. scoping problems within **let** and **letrec**

The following shows some typical examples of these errors and, where it may be unclear, discusses the problems encountered in each. The Scheme interpreter used in these examples, unless otherwise stated is SCM.

> (cdr 3)
The interpreter produces:
ERROR: cdr: Wrong type in arg1 3

The type system, however, is not much more instructive, as it complains:
Cannot be typed or your expression is too hard.

> (cons (cons (cons 3 '())))
Here, SCM is slightly more helpful:
ERROR: Wrong number of args to #<primitive-procedure cons>
in top level environment.

The type system, though, still reports:
Cannot be typed or your expression is too hard.

```
>(define buggy-remove
   (lambda (item ls)
      (cond
         ((null? ls) '())
```

((eq? item (car ls)) (buggy-remove (cdr ls)))
(else (cons car ls) (buggy-remove (cdr ls))))))

Here, no problem is reported by the interpreter, yet there is a serious problem: in the last line, the student has misplaced parentheses around the **car ls**. The last line should be **(else (cons (car ls) (buggy-remove (cdr ls)))))).** The type system produces both a warning, and an error message.
Implicit begin in cond expression.
ERROR: map: wrong type in arg2 item.

The error reported is a fatal error from the type system itself.

Another type of problem occurs in using Abstract Data Types (ADTs). Students often do not understand how to use these ADTs; they will access data members using inappropriate, or low-level, operations. An implementation for an example ADT *ratl* is given below.

```
(define make-ratl
  (lambda (num den)
    (list num den)))

(define numr
  (lambda (ratl)
    (car ratl)))

(define denr
  (lambda (ratl)
    (cadr ratl)))
```

This ADT is the first ADT students in the class encounter. The usual assignments require them to write various arithmetical operations for *ratls*. The most common programming error is for students to access the data members of a *ratl* using **car** or **cadr** instead of the selectors **numr** and **denr**. To force students to use the selectors, the implementation is sometimes changed or hidden from the students via loading a random implementation (e.g., dotted pair, Gödelized number, list with **denr** as the first element, procedural representation,etc.), yet students complain that using a random implementation is confusing, even though the instructor is merely trying to emphasize data hiding.

An additional tool to teach students to use data abstraction would be a type checker that could check students' code for low level selectors. Thus, a major goal of this project has been to implement a type checker that is sufficiently robust to handle ADTs, as well as provide better error information when expressions are improperly typed.

>(if #t 3 else 4)
The interpreter reports:
ERROR: no binding for else in the current environment.

The type system's message is similar:
Error: (unknown global name: else)

Here, the student is perhaps carrying over some syntax from another language; hence the use of **else**. Some other problems with **if** are listed below.

>(if 3 4)

In this example, the error is that the first argument does not evaluate to a boolean, and so the interpreter accepts it silently. The type system, though, reports an error:
Cannot be typed or your expression is too hard.

>(if #f (write "oops") 3)

Here, the second and third expressions have different types, yet is legal Scheme. The type system reports an error here, too.

Cannot be typed or your expression is too hard.

Students have similar problems with **cond**, as noted in the section on the subset of Scheme that is type-checked by this system.

Students also have scoping problems within a **let** or **letrec**. For example, the following code contains an invalid reference to the variable **x**.

```
>(let  ((x 3)
        (y x))
   (+ x y))
```

The **x** in the second line does not refer to the binding for **x** made in the first line. This is a very common error for beginning students to make, and the error message returned by the system is not always helpful.

ERROR: unbound variable: x

The message given by the type inference system is similar:
Error: (unknown global name: x)
Cannot be typed or your expression is too hard.

These seven types of problems are typical of beginning programming students, and a good type-checker should be able to at least point out these problems. Thus, the system described within this paper attempts to handle all of these problems, as well as some other, minor, points. It is clear, however, that the messages produced by the type system need improvement.

4.2. Demonstration of applicability

This section demonstrates that the type inference system is effective over its restricted domain. In most cases the type inference system correctly caught type errors, and correctly typed syntactically correct procedures. The procedure for checking this body of code was to type-check each file containing code, and examine the output. Procedures must be typed before they are referred to; thus mutually recursive procedures can cause problems; however, the mutually recursive procedures can be placed into a **letrec** expression, and then the individual procedures type-checked one at a time within the body of the **letrec**. A preliminary program, **type-check-file** has been developed that does this dependency handling, but it is limited in its usefulness as the size of files increases. The program reads in a file and attempts to resolve all dependencies; however, if there are more than one version of a program per file (for example, 3 versions of the **factorial** program), then the dependencies do not get resolved properly. Usually, though, a simple renaming takes care of this problem. Procedures of variable arity also cause significant problems: often the type system will die attempting to handle them. Also, since the programs below were from various sources, some of them relied on implementation-specific, or non-standard Scheme procedures. This problem was easily solved: definitions for the undefined helping procedures were included in the file, but not shown in the results table (they aren't included so as not to inflate the results). The results shown below were generated by running **type-check-file** over each of the files, with the three modifications mentioned: renaming of procedures, removal of procedures with variable arity, and inclusion of helping procedures.

The code from the first ten chapters of [SpF89] have been checked, as well as sample student code from exercises. Code from other sources was also examined. In particular, code from the first chapter of [AbS85] was examined. Since SICP uses nested **define**s, the report generated in the table was used setting the variable ***allow-nested-defines*** to **#t**, which then allowed the programs using nested **define**s to be desugared into programs using **letrec**. A final category of programs checked is a collection of polymorphic code by Gary Leavens. This was the most challenging set for the system as the interdependencies were more complex; it can be considered 'production level' code, and not just a set of small toy programs. It clearly shows the limitations of the type system.

The table below summarizes the results of using the type inference system over these bodies of Scheme code.

Source of Code	Number of procedures	Number of error message	Number of type errors	Percentage of procedures correct
Chapter 2, SAP	11	2	0	82%
Chapter 3, SAP	23	0	0	100%
Chapter 4, SAP	20	0	0	100%
Chapter 5, SAP	27	1	0	96%
Chapter 6, SAP	19	2	1	84%

Chapter 7, SAP	36	3	1	78%*(4 not included)
Chapter 9, SAP	23	1	3	74%*(2 not included)
Chapter 10, SAP	22	0	6	73%
Student code, SAP	16	3	0	81%
Chapter 1, SICP	101	5	0	95%
Polymorphic	26	5	4	65%
Totals	**324**	**22**	**16**	**86%**

There were few surprises while checking the code. Most problems occurred with recursive types, such as trees, or with **make-vector**, which will be discussed in detail later.

Chapter 2 of [SpF89] produced 4 errors: three of these were problems with procedures taking advantage of Scheme's using any non-**#f** value to mean true. The only other problem was with a procedure that performed *ad hoc* polymorphism, and, thus, is beyond the scope of our system. The code for the procedure is shown below.

```
(define describe
   (lambda (s)
      (cond
         ((null? s) (quote '()))
         ((number? s) s)
         ((symbol? s) (list 'quote s))
         ((pair? s) (list 'cons (describe (car s)) (describe (cdr s))))
         (else s))))
```

Chapter 3 produced no errors. Even though much of the procedures deal with the ADT *ratls*, all the procedures built on top of the ADT successfully checked.

Almost all of the problems in chapter 4 were caused by procedures that operate over trees producing error messages. An example is the procedure **remove-all**:

```
(define remove-all
   (lambda (item ls)
      (cond
         ((null? ls) '())
         ((equal? (car ls) item) (remove-all item (cdr ls)))
         ((pair? (car ls))
           (cons (remove-all item (car ls))
                 (remove-all item (cdr ls))))
         (else (cons (car ls) (remove-all item (cdr ls)))))))
```

The flat version of **remove-all** can be handled; i.e., without the case where **ls** is a *pair*. But there are no facilities for recursive types, such as trees, in the type system. The problem also came up later when checking the polymorphic code of Gary Leavens.

In chapter 5, the only problem is with a deep-recursive procedure, as in chapter 4.

In chapter 6, all of the problems were caused by interactive procedures: some of the interactive procedures were fine; however, some did not like being able to handle different kinds of input via **read**. For example, the following code produces an error:

```
(define interactive-square
   (lambda ()
      (let ((val (read)))
         (if (eq? val 'done)
            (writeln "Thanks for playing")
            (begin
               (writeln "The square of " val " is " (* val val))
               (interactive-square))))))
```

The problem occurs because **read** returns both a *symbol* (i.e., 'done) and a *number*.

Chapter 7 had problems with trees, also, and our inability to infer types for procedures with variable arity also produced the majority of these errors. A typical example for problems with variable arity is this: **(define list (lambda args args))**.

Chapters 9 and 10 produced errors almost every time **vector-generator** was used. This is simple to understand as **vector-generator** uses **make-vector**, which produces a vector of type *datum* when called with no fill value. These, though, were the primary problems in these chapters. The code for **vector-generator** causes a problem as it initially creates a vector of type *datum* using **make-vector**, and then updates it according to the **gen-proc**. However, since **vector-set!** does not allow non-homogenous vectors to be created, all vectors made with **vector-generator** are of type *datum*. Thus chapters 9 and 10 have few error messages produced, but several incorrect types. The code for **vector-generator** is given here.

```
(define vector-generator
   (lambda (gen-proc)
      (lambda (size)
         (let ((vec (make-vector size)))
            (letrec ((loop (lambda (i)
                           (if (< i size)
                              (begin
                                 (vector-set! vec i (gen-proc i))
                                 (loop (add1 i))))))
               (loop 0))
            vec))))
```

The student's code consisted of 16 procedures (all correct Scheme code) that were solutions to exercises from the first ten chapters of [SpF89]. The problems here, as mentioned above, were primarily with **vector-generator** and procedures that take a variable number of arguments.

The code from the first chapter of [AbS85] was the most successfully typed code set. Essentially no errors were produced after some simple transformations were done; i.e.,

renaming several versions of the same procedure, and adding definitions for some system-specific procedures (e.g., **1+** and **1-**).

The final group of code checked consists of code by Gary Leavens. The problems encountered here were from a recursive ADT, much as from the tree code in chapter 4. The example here is a type *type*, which can be a symbol or a list of (type -> type), much as types can be expressed in the system developed in this project.

From this data it is clear that a large portion of Scheme code (i.e., the subset we have delimited) can be successfully type-checked using our system. The major limitation is in the use of recursive types.

4.3. Related works

In this section we attempt to place out type inference sytem in context with other type systems for Scheme, as well as other functional languages. We will take a brief look at PLEAT [Cur90], STYLE [Lin93], Soft-Scheme [WrC93], and SPS [Wan89], with the emphasis in the section being on STYLE and Soft-Scheme. We will look at three main areas: domain of the type systems, representation of types, and complexity of code.

The work of Curtis [Cur90] provides an example of a type-system for a small, functional language, PLEAT, much in the style of Scheme. However, the types produced by the system are far too complex for beginning students to handle. For example, the procedure **sum**, which takes a list of numbers and returns the sum of the list has type *(-> (list number) number)* in our type system, but in Curtis', it has the type

$$\forall\alpha.\,(rec\;\beta.\,[\,Empty:\alpha,\;NonEmpty:\langle hd:\mathrm{int},\,tl:\beta\rangle\,]\;\rightarrow\;\mathrm{int})$$

While to someone familiar with Curtis' presentation of recursion and his type for **cons**, this is understandable, it is clearly beyond beginning students' ability to use. However, his type system is quite rich and complete. It is just not a good fit for our goal of helping students understand types by showing them how typing problems in their code are usually errors.

The Semantic Prototyping System (SPS) of Wand is also instructive. While the types produced by it are more usable for students, there is a serious problem in using it: it requires that students enter in types for their procedures, and then attempts to perform a unification with the type defined by the student, and the type of the procedure declared. Using this on top of existing code proves difficult without employing macros to translate procedures with their types into this language. The only other option would be to teach the students the syntax of SPS, in addition to the regular syntax of Scheme. Below shows code, with the type, for a procedure **myzero?**, along with the equivalent type in our system.

```
(define-checked myzero?
   (-> (seq int) bool)
   (lambda (n) (eq? n 0))))
```

The equivalent type in our system is *(-> (T) boolean)*. Thus, the notation for types is not very different, yet the overhead of **define-checked**, and requiring students to enter

presumed types makes using the system burdensome. The concept, though, does have merit. As part of the introductory course, students are required to include types for all procedures. However, requiring these types to be correct puts too much pressure on them.

There are two other major problems with SPS: it does not correctly implement **let** and it has no facilities for ADTs. While the first is not crucial to the use of a type system (until students reach higher-order procedures in Scheme), the second causes SPS to be rejected as a possibility for our use. Thus, while SPS does meet some of our goals, it is not a proper fit for our course.

A type system that is very close to ours is Christian Lindig's STYLE [Lin93]. It offers a type system that operates over the entire specification of Scheme, as well as provides a solid type system. However, at the time of this writing, we have been unable to locate a copy of the implementation, although the report we have states that the code will be soon made available via anonymous ftp. Thus, any data we have on STYLE relies on the technical reports, and we have been unable to compare it directly with our system. Another drawback of STYLE is its complex type system. The types produced are not very palatable. For example, a version of **member?**, has type *(A_nv (B_nv . C_nv) => bool)* in Lindig's system, while it has type
(-> (T (list T)) boolean) in ours. Note that the comparison is made on the basis of the type given in [Lin93], which does not provide the source for **member?**; it is assumed that the implementation is something resembling:

```
(define member?
   (lambda (item ls)
      (cond
         ((null? ls) #f)
         ((equal? item (car ls)) #t)
         (else (member? item (cdr ls))))))
```

In his paper, Lindig provides some results from type-checking code from [AbS85], and from that, we see that his system appears quite practical. However, since we have been unable to obtain a copy, we cannot make any accurate comparisons of our two systems at this time.

The last system examined is Wright's and Cartwright's Soft Scheme [WrC93]. Soft Scheme infers types, and, instead of producing error messages on untypable expressions, inserts run-time checks. The system covers all of R4RS Scheme and produces readable, usable types. This system is very poweful, and has been shown to perform very well. For our purposes, there are only two problems with the system: it is too complex for students to comprehend the implementation, and it doesn't handle ADTs in the same manner that [SpF89] does. It does, however, have facilities for defining and using structures. These, however, are more suited for more advanced programmers, not students learning about ADTs. Its handling of recursive types and intersection types is based on the work of Fagan [Fa90] and is richer, yet more complicated, than ours.

The implementation of Soft Scheme is available via ftp from the Scheme repository (ftp.cs.indiana.edu in pub/scheme-repository/imp). A working implementation consists of

nearly 7000 lines of Scheme code, using many of the more advanced constructs of Scheme (namely **call/cc** and **extend-syntax**). Thus, the system is far more complex that what students could be expected to look over in their first semester programming. However, this author has found Soft Scheme to be quite nice, and had the system been available at the beginning of this project, the facilities for ADTs might have been merely built on top of Soft Scheme.

4.4. Theoretical Basis

The theoretical basis for this work can be found primarily in the papers of Cardelli and Milner. The type-checking algorithm is essentially the **j** algorithm of Milner, with a few additions as noted in the sections on *datum*, *poof*, *and-types*, and ....

Milner, of course, provides the basic type-checking algorithm, as well as the underlying notions of type-checking polymorphic functions. Milner's work is built upon the framework of Hindley [Hin69].

The paper by Cardelli [Car87] presents the basis for polymorphic type-checking, as well as provides some basic type-expressions and type-inference rules. The goal for the system implemented here is to have the same functionality as the system described in [Car87], as well as provide for ADTs and some exception-handling. However, unions of types have been simplified as noted in the section on *datum*. Thus we have implemented Cardelli's ideas for polymorphic type inference in Scheme, and provided a simple, controlled use of subtyping, along with some facilities for creating and using higher-order types.

Curtis [Cur90] introduces the use of quantified variables, and presents his version of a type-inference system for a language similar to Scheme called PLEAT. As noted in the previous section, however, his type system is too complicated for our needs.

The work by Pierce [Pie91], Reynolds [Rey88], and Coppo, Dezani-Ciancaglini, and Venneri [CD80], provide additional insight into *and-types*, even though they are only handled in a primitive fashion in this work.

5. FUTURE DIRECTIONS

All of the syntactic features used to create the system are covered in the introductory programming course, except for a few functions on file handling. Thus, not only can students use the system to type check their code, but they can also modify it on their own and experiment with different approaches. Also, all of the code is compliant with R4RS, and thus is completely portable: no special, implementation dependent features have been used. This system has been tested under SCM, Chez Scheme, and PCS and runs with no modifications. The only procedure not in R4RS compliant Scheme is **file-exists?**, which exists on all three of the above implementations.

Further goals for this research include extending it to include all features of Scheme, not just a selected subset. However, the inclusion of more features might impair the portability of the system; thus any modifications will be made with care. While the addition of record types would be nice, their inclusion could impair the portability of the system. However, depending on SLIB to provide the appropriate facilities would aid the portability of this kind of extension. Thus, if SLIB where already installed in the Scheme

system, no other modifications would be necessary. This would be especially helpful for handling macros, as SLIB currently provides a portable version of **extended-syntax**.

6. CONCLUSION

In conclusion the project has achieved most of its goals. The type-inference system is sufficiently adequate and robust to handle the needs of beginning students. And it is simple enough for them to use, understand, and even modify. The simplicity of the type system allows users to experiment with different types for different procedures and allow great flexibility in how much polymorphism is allowed (i.e., via *and-types*). However, it would be nice to cover all of the features used in Scheme, not just a restricted subset. Thus, the further goal of this work is to incorporate all of R4RS.

BIBLIOGRAPHY

[AbS85] Harold Abelson, Gerald Sussman and Julie Sussman, Structure and Interpretation of Computer Programs, 13th edition, The MIT Press, Cambridge, Massachusetts.

[Car87] Luca Cardelli, Basic Polymorphic Typechecking, Science of Computer Programming, 8,2 (April 1987).

[CR91] William Clinger and Jonathan Rees (eds.), Revised^4 Report on the Algorithmic Language Scheme, 1991.

[CD80] Mario Coppo, M. Dezani-Ciancaglini, and B. Venneri, Principal type schemes and lambda calculus semantics, in To H. B. Currry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, pp. 535-560, New York, Academic Press, 1980.

[Cur90] Pavel Curtis, Constrained Quantification in Polymorphic Type Analysis, Xerox PARC Technical Report CSL-90-1, February 1990.

[DH94] Hsianlin Dzeng and Christopher T. Haynes, Type Reconstruction for Variable-Arity Procedures, 1994, to be published.

[Fa90] M. Fagan, Soft Typing: An Approach to Type Checking for Dynamically Typed Languages, PhD thesis, Rice University, 1992.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic, Transactions of the American Mathematical Society, Volume 146, December 1969, pp. 29-60.

[Lin93] Christian Lindig, STYLE: A Practical Type Checker for Scheme, Technische Universität Braunschweig, Informatik-Bericht Nr. 93-10, October 1993.

[Mil78] Robin Milner, A Theory of Type Polymorphism in Programming, Journal of Computer and System Sciences, 17 (1978), 348-375.

[Pie91] Benjamin C. Pierce, Programming With Intersection Types, Union Types, and Polymorphism, CMU-CS-91-106, 1991.

[Rey88] John Reynolds, Preliminary design of the programming language Forsythe, Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[SpF89] George Springer and Daniel P. Friedman, Scheme and the Art of Programming, MIT Press and McGraw-Hill, 1989.

[Wa89] Mitchell Wand, Semantic Prototyping System (SPS) Reference Manual, Version 1.4 (Chez Scheme), Northeastern University, 1989.

[WrC93] Andrew K. Wright and Robert Cartwright, A Practical Soft Type System for Scheme, Rice University Technical Report, TR93-218, December 6, 1993.